DOCUMENT PREPARATION SYSTEMS
J. Nievergelt, G. Coray, J.D. Nicoud, A.C. Shaw (eds.)
© North-Holland Publishing Company, 1982

Choosing Better Line Breaks*

MICHAEL F. PLASS
Xerox Palo Alto Research Centers
3333 Coyote Hill Road
Palo Alto, California 94304, U.S.A.

DONALD E. KNUTH
Computer Science Department
Stanford University
Stanford, California 94305, U.S.A.

This article discusses a new approach to the problem of dividing the text of a paragraph into lines of approximately equal length. Instead of simply making decisions one line at a time, the method considers the paragraph as a whole, so that the final appearance of a given line might be influenced by the text on succeeding lines. A system based on three simple primitive concepts called 'boxes', 'glue', and 'kerfs' provides the ability to deal satisfactorily with a wide variety of typesetting problems in a unified framework, using a single algorithm that determines optimum breakpoints. This algorithm avoids backtracking by a judicious use of the techniques of dynamic programming. Extensive computational experience confirms that the approach is both efficient and effective in producing high-quality output.

INTRODUCTION

The software of a document preparation system may be divided into several parts, depending upon the function performed: user interface (getting the input from the user, perhaps interactively), system interface (conversion between the internal and external representations of the documents and font descriptions), control (getting the different parts of the system to work together in a coherent way), and translation (a non-trivial conversion from one representation to another). Underlying all of these kinds of software are basic algorithmic building blocks. The most fundamental of these pervade the software at all levels; examples are various kinds of table lookup algorithms, and algorithms for manipulation of linked data structures. Others are applicable only to some particular aspect of the system as a whole. Hyphenation algorithms would fall into this latter class; although hyphenation could, in principal, be treated as a table lookup problem, it is usually advantageous to summarize the hyphenation rules in some more compact form, and then to use a table of exceptions. Another example would be algorithms for parsing, i.e., for converting from a flat, string-oriented input representation to a structured internal representation; the lessons learned from compiler construction can be helpful in this area. There are special purpose algorithms for converting from one description of a font (say a polygonal outline) to another (say a bit matrix). The chief purpose of this report is to describe a particular special-purpose algorithm, for the problem of breaking paragraphs into lines.

Before we plunge into the details of the line-breaking problem, we should consider how it fits into the system as a whole. One of the most important aspects of a document preparation system is the *model* it presents to the user. A model will ideally have the following properties:

- *Simplicity* — a simple model can be concisely explained and easily understood in terms of just a few primitive concepts. A user who understands the underlying model and has a particular task in mind will be able to figure out how to do it without searching a lengthy manual to discover the right 'feature' to use.

* This research was supported in part by the National Science Foundation under grants IST-7921977 and MCS-7723738; by Office of Naval Research grant N00014-76-C-0330; by the IBM Corporation; by Addison-Wesley Publishing Company; and by the Xerox Corporation. 'TEX' and 'Tau Epsilon Chi' are registered trademarks of the American Mathematical Society. This report contains substantial excerpts from an earlier paper appearing in Software—Practice and Experience[11], © John Wiley & Sons, Ltd., used by permission.

- *Generality* — a general model allows the user create any document that can be produced with the available hardware.
- *Naturalness* — a natural model conforms well with the user's notion of how the document is put together.

For example, one possible model of a document is simply a list of items, where each item specifies a font, a character, and a position on the page. This model is certainly simple and quite general, but for the user trying to prepare a document consisting mostly of text, it is not at all natural. It is, however, quite an appropriate model for the output device to present to the typesetting software, since the final internal representation of the document will generally be easy to convert to this form (easy for a computer, that is).

Another possible model would be the 'typewriter model', in which the user just gives a string of characters, perhaps including characters to start a new line or a new page. This model is much more natural for most applications, and also very simple, but it is lacking in generality. It can serve, however, as the starting point for a more general model, at the cost of some of its simplicity and naturalness.

A case can be made that the model that is finally presented to the user should sometimes sacrifice some simplicity and/or generality to make the user interface more natural, or to achieve some other goal. Perhaps a goal is to separate the task of the author from that of the document designer; then the model presented to the author should not be fully general, or else the intentions of the document designer could be circumvented. Or it might be hard for a novice user to figure out how to apply a simple model to obtain some common desired result; in such a case a long list of different features might be preferable. But if the user's model can be expressed completely in terms of a simpler, more general internal model, many advantages can be obtained. In the first example, the document designer can express the document design in terms of the internal model, and so obtain the necessary generality. In the second example, if the features used by the novice are implemented by building on top of a simple, general model, they will be just as useful as if they had been implemented on a case-by-case basis. The advanced user, on the other hand, will not be limited to this set of features, but will have access to the power and flexibility obtained through the combination of primitive functions.

The remainder of this article will concentrate on one particular model for the contents of a document, and one particular algorithm for operating on this model. The algorithm, as mentioned above, is aimed at solving the problem of breaking a paragraph into lines, and in this article we will develop only those aspects of the model that are relevant to this problem.

The line-breaking problem is informally called the problem of 'justification', since it is the 'J' of 'H & J' (hyphenation and justification) in today's commercial composition and word-processing systems. However, this tends to be a misnomer, because printers have traditionally used justification to mean the process of taking an individual line of type and adjusting its spacing to produce a desired length. Even when text is being typeset with ragged right margins (therefore 'unjustified'), it needs to be broken into lines of approximately the same size. The job of adjusting spaces so that left and right margins are uniformly straight is comparatively laborious when one must work with metal type, so the task of typesetting a paragraph using last century's technology was conceptually a task of justification; nowadays, however, it is no trick at all for computers to adjust the spacing as desired, so the line-breaking task predominates the work. This shift in relative difficulty probably accounts for the shift in the meaning of 'justification'; we shall use the term 'line breaking' in this chapter to emphasize the fact that the central problem of concern here is to find breakpoints.

The traditional way to break lines is analogous to what we ordinarily do when using a typewriter: A bell rings (at least conceptually) when we approach the right margin, and at that time we decide how best to finish off that line, without looking ahead to see where the next line or lines might end. Once the typewriter carriage has been returned to the left margin, we begin afresh without needing to remember anything about the previous text except where the

previous line starts. Thus we don't have to keep track of many things at once; such a system is ideally suited to human operation, and it also leads to simple computer programs.

Book printing is different from typing primarily in that the spaces are of variable width. Traditional practice has been to assign a minimum and maximum width to interword spaces, together with a normal width representing the ideal situation. The standard algorithm for line breaking (see, for example, Barnett[1], page 55) then proceeds as follows: Keep appending words to the current line, assuming normal spacing, until reaching a word that does not fit. Break after this word, if it is possible to do so without compressing the spaces to less than the minimum; otherwise break before this word, assuming it is possible to do so without expanding the spaces to more than the given maximum. Otherwise hyphenate the offending word, putting as much of it on the current line as will fit; if no suitable hyphenation points can be found, this may result in a line whose spaces exceed the given maximum.

There is no need to confine computers to such a simple procedure, since the data for an entire paragraph is generally available in the computer's memory. Experience has shown that significant improvements are possible if the computer takes advantage of its opportunity to 'look ahead' at what is coming later in the paragraph, before making a final decision about where any of the lines will be broken. This not only tends to avoid cases where the traditional algorithm has to resort to wide spaces, it also reduces the number of hyphenations. In other words, line breaking decisions provide another example of the desirability of 'late binding' in computer software.

One of the principal reasons for using computers in typesetting is to save money, but at the same time we don't want the output to look cheaper. A properly programmed computer should, in fact, be able to solve the line-breaking problem better than a skilled typesetter could do in a reasonable amount of time (unless we give this person the liberty to change the wording in order to obtain a better fit). For example, Duncan[2] studied the interword spacing of 958 lines that were manually typeset by a "most respectable publishers' printer" that he chose not to identify by name, and he found that nearly 5% of the lines were quite loosely set; the spaces on those lines exceeded 10 units (i.e., $\frac{10}{18}$ of an em), and two of the lines even had spaces exceeding 13 units. We shall see later that a good line-breaking algorithm can do better than this.

Besides the avoidance of hyphens and wide spaces, we can improve the traditional line-breaking method by keeping the spaces nearly equal to the normal size, so that they rarely approach the minimum or maximum limits. We can also try to avoid rapid changes in the spacing of adjacent lines; we can make special efforts not to hyphenate two lines in a row, and not to hyphenate the second-last line of a paragraph; and so on. Given any mathematical way to rate the quality of a particular choice of breakpoints, we can ask the computer to find breakpoints that optimize this function.

But how is the computer to solve such a problem efficiently? When a given paragraph has n optional breakpoints, there are 2^n ways to break it into lines, and even the fastest conceivable computers could not run through all such possibilities in a reasonable amount of time. In fact, the problem sounds suspiciously like the infamous bin-packing problem, which is well known to be NP-complete[3]. Fortunately, however, each line is to consist of contiguous information from the paragraph, so the line-breaking problem is amenable to the techniques of discrete dynamic programming[4,5]; this means there is a reasonably efficient way to attack it. We shall see that the optimum breakpoints can be found in practice with only about twice as much computation as needed by the traditional algorithm; the new method is sometimes even faster than the old, when we consider the time saved by not needing to hyphenate so often. Furthermore the new algorithm is capable of doing other things that are useful in laying out pages, such as setting a paragraph one line longer or one line shorter than normal.

FORMULATING THE PROBLEM

For our purposes, a *paragraph* is a sequence $x_1, x_2, ..., x_m$ of m items, where each individual item x_i specifies a *box*, *glue*, or a *kerf*.

- A box refers to something that is to be typeset: either a character from some font of type, or a black rectangle such as a horizontal or vertical rule, or something built up from several characters such as an accented letter or a mathematical formula. The contents of a box may be extremely complicated, or extremely simple; the line-breaking algorithm does not peek inside a box to see what it contains, so we may consider the boxes to be sealed and locked. As far as we are concerned, the only relevant thing about a box is its *width*. The width of a box may be zero, and in fact it may even be negative, athough negative widths must be used with care and understanding. We use the notation 'box(w)' to denote a box of width w.
- Glue refers to blank space that can vary its width in specified ways; it is an elastic mortar used between boxes in a typeset line. A glue specification x includes just three numbers of relevance to line breaking: $width(x)$ is the 'ideal' or 'normal' width; $stretch(x)$ is the 'stretchability'; $shrink(x)$ is the 'shrinkability'.

  For example, the amount of space between two words in a line often contains the glue specification $width(x) = \frac{1}{3}$ em, $stretch(x) = \frac{1}{6}$ em, $shrink(x) = \frac{1}{9}$ em, where one em is the set size of the type being used (approximately the width of an uppercase 'M' in classical type styles). The actual amount of space occupied by this glue can be adjusted when the line containing it is justified to some desired width; if the normal width is too small the adjustment is proportional to $stretch(x)$, and if the normal width is too large the adjustment is proportional to $shrink(x)$. A glue item with width w, stretch y and shrink z will be denoted by 'glue(w, y, z)'. Notice that, for the purposes of actually choosing the line breaks, a box can be considered to be equivalent to a glue item with zero stretch and shrink; we will maintain the distinction, however, because there is a big difference between boxes and glue in othe parts of the document system (otherwise it would just produce blank sheets of paper).
- A kerf describes a place in the paragraph where a break is possible, and specifies what is supposed to go into the lines just before and after the break (if the paragraph is broken at this place), or what is used in place of the break (if no break happens here). Specifically, the kerf specification x contains three lists of boxes and glue: $prebreak(x)$ contains the boxes and glue to be appended onto the end of the line just before the break; $postbreak(x)$ contains the boxes and glue to be appended onto the beginning of the line immediately following the break; $join(x)$ contains the boxes and glue to be inserted into the line in case no break is chosen at this kerf. For the purposes of line breaking, the exact composition of these lists is not important, but merely the sums of the widths, stretches, and shrinks; we will continue to think of them as lists, however, to keep the model closer to reality. In addition to the three lists, the kerf specification may include a number $penalty(x)$ that quantifies how undesirable a break at x would be, and a boolean flag $hyph(x)$ that tells whether or not a break at this kerf is to be counted as a hyphenation. The hyphenation flag is useful in avoiding breaks that would cause consecutive lines to end with hyphens.

Any fixed unit of measure can be used for the dimensions *width*, *stretch*, and *shrink*; TEX uses printers' points, which are slightly less than $\frac{1}{72}$ inch. In this paper we shall specify all dimensions in term of *machine units* equal to $\frac{1}{18}$ em, assuming a particular size of type, since the widths turn out to be integer multiples of this unit in many cases; the numbers in our examples will be as simple as possible when expressed in terms of machine units.

In order to better internalize the abstract concepts embodied in the definitions of boxes, glue, and kerfs, let us think of how they can be applied to the typesetting of straight text material, such as a newspaper article or a short story. A typesetting system will put such a paragraph

into the abstract form we want in the following way:

(1) If the paragraph is to be indented, the first item will be an empty box whose width is the amount of indentation; alternatively, it could be a glue item with the requisite width and with zero stretch and shrink.

(2) Each word of the paragraph becomes a sequence of boxes for the characters of the word, including punctuation marks that belong with that word.

(3) Each interword space becomes a kerf with *prebreak* and *postbreak* empty, *penalty* zero, *hyph* false, and *join* containing a glue item corresponding to the recommended spacing conventions of the fonts of type in use. The glue might be different in different contexts; for example, TEX will make the glue specifications following punctuation marks slightly different from the normal interword glue.

(4) At each place within the words where a hyphenation is permissible, there is a kerf with *postbreak* and *join* empty, *penalty* nonzero, *hyph* true, and *prebreak* containing a box with a hyphen inside.

(5) Explicit hyphens and dashes in the text will be followed by kerfs with all three lists empty, *hyph* true, and with the penalty as appropriate.

(6) At the very end of the paragraph, a kerf z is appended, with *prebreak*(z) containing zero-width glue with zero shrink and infinite (or very large) stretch, *postbreak* empty, *penalty* zero, *hyph* true (to try to avoid a hyphenation in the next-to-last line), and *join* containing an infinite width item, so as to force a break at this kerf.

This translation of the paragraph will cause it to be justified, with both margins straight. If we wanted it to be set with a ragged right margin, then in (3) we would use a fixed width glue in *join* and a zero-width glue with some arbitrary non-zero amount of stretch in *prebreak*. It would also be necessary to include this same glue after the hyphen in the *prebreak* of (4) and in the *prebreak* of (5). We will see later that the box-glue-kerf model can also be used to solve many of the more esoteric typesetting problems that come up in practice.

It should be noted that the translation into boxes, glue, and kerfs would normally be done entirely by the typesetting program, given the text of the paragraph and some description of the style desired. The strength of the model is that the same line breaking algorithm can be used to meet the requirements of many different styles, so that the advantages of a good line breaking algorithm can be made to apply uniformly to them all.

Desirability Criteria

According to this definition of line breaking, there are 2^n ways to break a paragraph into lines, if the paragraph has n legal breakpoints that aren't forced. For example, let's consider the paragraph of Figure 1, which is taken from Grimm's Fairy Tales[6]. There are 129 kerfs in this paragraph, not counting the forced break at the end, and so it can be broken into lines in 2^{129} ways, a number that exceeds 10^{38}. But of course most of these choices are absurd, and we need to specify some criteria to separate acceptable choices from the ridiculous ones. For this purpose we need to know (a) the desired lengths of lines, and (b) the lengths of lines corresponding to each choice of breakpoints, including the amount of stretchability and shrinkability that is present. Then we can compare the desired lengths to the lengths actually obtained.

We shall assume that a list of desired lengths $l_1, l_2, l_3, \ldots$ is given; normally these are all the same, but in general we might want lines of different lengths, as when fitting text around an illustration. Let the actual length of the jth line be denoted by L_j; this is calculated in the obvious way by adding together the widths in the list formed by concatenating the *postbreak* of the kerf at the beginning of the line, the boxes, glue, and *join*s of the kerfs included in the line, and the *prebreak* of the kerf at the end of the line. In a similar fashion we find the total stretchability Y_i and the total shrinkability Z_i. Now we can compare the actual length L_j to

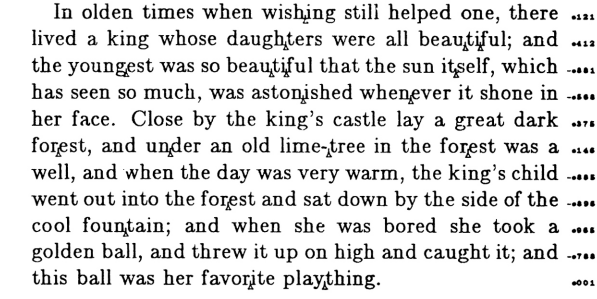
In olden times when wishing still helped one, there ..121
lived a king whose daughters were all beautiful; and ..112
the youngest was so beautiful that the sun itself, which ..001
has seen so much, was astonished whenever it shone in ..000
her face. Close by the king's castle lay a great dark ..071
forest, and under an old lime-tree in the forest was a ..100
well, and when the day was very warm, the king's child ..000
went out into the forest and sat down by the side of the ..000
cool fountain; and when she was bored she took a ..000
golden ball, and threw it up on high and caught it; and ..700
this ball was her favorite plaything. ..001

Figure 1. An example paragraph that has been typeset by the 'first-fit' method. Small triangles show permissible places to divide words with hyphens; the adjustment ratio for spaces appears at the right of each line.

the desired length l_j by seeing if there is enough stretchability or shrinkability to change L_j into l_j; we define the *adjustment ratio* r_j of the jth line as follows:

If $L_j = l_j$ (a perfect fit), let $r_j = 0$.

If $L_j < l_j$ (a short line), let $r_j = (l_j - L_j)/Y_j$, assuming that $Y_j > 0$; the value of r_j is undefined if $Y_j \leq 0$ in this case.

If $L_j > l_j$ (a long line), let $r_j = (l_j - L_j)/Z_j$, assuming that $Z_j > 0$; the value of r_j is undefined if $Z_j \leq 0$ in this case.

Thus, for example, $r_j = \frac{1}{3}$ if the total stretchability of line l is three times what would be needed to expand the glue so that the total line length would change from L_j to l_j.

According to this definition of adjustment ratios, the jth line can be justified by letting the width of all glue items x_i within that line be

$$\begin{aligned} &width(x_i) + r_j\, stretch(x_i), && \text{if } r_j \geq 0; \\ &width(x_i) + r_j\, shrink(x_i), && \text{if } r_j < 0. \end{aligned}$$

For if we add up the total width of that line after such adjustments are made, we get either $L_j + r_j Y_j = l_j$ or $L_j + r_j Z_j = l_j$, depending on the sign of r_j. This distributes the necessary stretching or shrinking by amounts proportional to the individual glue components $stretch(x_i)$ or $shrink(x_i)$, as desired.

For example, the small numbers at the right of the individual lines in Figure 1 show the values of r_j in those lines. A negative ratio like —.881 in the third line means that the spaces in that line are narrower than their ideal size; a fairly large positive ratio like .965 indicates a very 'loose' fit.

Although there are 2^{129} ways to break the paragraph of Figure 1 into lines, it turns out that only 49 of these actually will result in breaks whose adjustment ratios do not exceed 1 in absolute value; this means that the spaces between words after justification will lie between $width(x_i) - shrink(x_i)$ and $width(x_i) + stretch(x_i)$. Furthermore, only 30 of these 49 ways to make 'nice' breaks will do so without introducing hyphens. One of these ways appears in Figure 1; another way, slightly better, is obtained by moving 'the' from the eighth line down to the ninth.

Our main goal is to find a way to avoid choosing any breakpoints that lead to lines in which the words are spaced very far apart, or in which they are very close together, because such lines are distracting and harder to read. We might therefore say that the line-breaking problem is to find breaks such that the adjustment ratio in each line never exceeds one in absolute value, with the minimum number of hyphenations subject to this condition. Such an approach was taken by Duncan et al.[7] in the early 1960s, and they obtained fairly good results. However, this condition depends only on the values of $width(x_i) - shrink(x_i)$ and $width(x_i) + stretch(x_i)$, not on $width(x_i)$ itself, so it does not use all the degrees of freedom

In olden times when wishing still helped one, there ..121
lived a king whose daughters were all beautiful; and ..112
the youngest was so beautiful that the sun itself, which ..001
has seen so much, was astonished whenever it shone ..111
in her face. Close by the king's castle lay a great dark ..001
forest, and under an old lime-tree in the forest was a ..110
well, and when the day was very warm, the king's child ..001
went out into the forest and sat down by the side of ..100
the cool fountain; and when she was bored she took a ..001
golden ball, and threw it up on high and caught it; ..002
and this ball was her favorite plaything. ..001

Figure 2. The paragraph of Figure 1 when the 'best-fit' method has been used to find successive breakpoints.

present in our data. Furthermore, such stringent conditions may not be possible to achieve; for example, if each line of our example were to be 418 units wide, instead of the present width of 421 units, there would be no way to set the text of Figure 1 without having at least one very tight line or loose line, i.e., a line with the magnitude of the adjustment ratio greater than one.

We can do a better job of line breaking if we deal with a continuously varying criterion of quality instead of the yes/no tests on the magnitude of the adjustment ratios. Let us therefore give a quantitative rating of the *badness* of the jth line by finding a formula that is nearly zero when $|r_j|$ is small but grows rapidly when $|r_j|$ takes values exceeding 1. Experience with TEX has shown that good results are obtained if we define the badness of line j as follows:

$$\beta_j = \begin{cases} \infty, & \text{if } r_j \text{ is undefined or } r_j < -1; \\ 100|r_j|^3, & \text{otherwise.} \end{cases}$$

Thus, for example, the individual lines of Figure 1 have badness ratings that are approximately equal to 0, 7, 68, 18, 5, 0, 69, 72, 90, 49, 0, respectively. Note that a line is considered to be 'infinitely bad' if $r_j < -1$; this means that glue will never be shrunk by more than its *shrink* value. However, values of $r_j > 1$ are only finitely bad, so they will be permitted if there is no better alternative.

A slight improvement over the method used to produce Figure 1 leads to Figure 2. Once again each line has been broken without looking ahead to the end of the paragraph and without going back to consider previous choices, but this time each break was chosen so as to minimize the 'badness plus penalty' of that line. In other words, when choosing between alternative ways to end the jth line, given the ending of the previous line, we obtain Figure 2 if we take the minimum possible value of $\beta_j + \pi_j$; here β_j is the badness as defined above, and π_j is the amount of penalty associated with the kerf at the end of the line. In our example, the penalty is 50 at a hyphenation and zero everywhere else. Figure 2 improves on Figure 1 by moving words down from lines 4, 8, and 10.

The method that produces Figure 1 might be called the 'first-fit' algorithm, and the corresponding method for Figure 2 might be called the 'best-fit' algorithm. We have seen that best-fit is superior to first-fit in this particular case, but other paragraphs can be contrived in which first-fit finds a better solution; so a single example is not sufficient to decide which method is preferable. In order to make an unbiased comparison of the methods, we need to get some statistics on their 'typical' behavior. Therefore 300 experiments were performed, with line widths ranging from 350 to 649 in unit steps; although the text for each experiment was the same, the varying line widths made the problems quite different, since line-breaking algorithms are quite sensitive to slight changes in the measurements. The 'tightest' and 'loosest' lines in each resulting paragraph were recorded, as well as the number of hyphens introduced, and the

In olden times when wishing still helped one, there ..121
lived a king whose daughters were all beautiful; and ..112
the youngest was so beautiful that the sun itself, which ..1001
has seen so much, was astonished whenever it shone ..1004
in her face. Close by the king's castle lay a great dark ..1003
forest, and under an old lime-tree in the forest was ..703
a well, and when the day was very warm, the king's ..101
child went out into the forest and sat down by the side ..1014
of the cool fountain; and when she was bored she took ..1001
a golden ball, and threw it up on high and caught it; ..104
and this ball was her favorite plaything. ..001

Figure 3. This is the 'best possible' way to break the lines in the paragraph of Figures 1 and 2, in the sense of fewest total 'demerits' as defined in the text.

comparisons came out as follows:

	min j	max j	hyphens
first-fit $<$ best-fit	69%	35%	12%
first-fit $=$ best-fit	26%	50%	77%
first-fit $>$ best-fit	5%	15%	11%

Thus in 69% of the cases, the minimum adjustment ratio in the lines typeset by first-fit was less than the corresponding value obtained by best-fit; the maximum adjustment ratio in the first-fit lines was less than the maximum for best-fit about 35% of the time, etc. We can summarize this data by saying that the first-fit method usually typesets at least one line that is tighter than the tightest line set by best-fit, and it also usually produces a line that is as loose or looser than the the loosest line of best-fit. The number of hyphens is about the same for both methods, although a more detailed study of the experimental data shows that the superiority of best-fit is especially pronounced in the cases where the lines are rather narrow.

We can actually do better than both of these methods by finding an 'optimum' way to choose the breakpoints. For example, Figure 3 shows how to improve on both Figures 1 and 2 by making line 6 a bit looser, therefore avoiding a rather tight 7th line and a fairly loose 10th line. This pattern of breakpoints was found by an algorithm that will be discussed in more detail below. It is globally optimum in the sense of having fewest total 'demerits' over all choices of breakpoints, where the demerits assessed for the jth line are computed by the formula

$$\delta_j = \begin{cases} (1+\beta_j+\pi_j)^2+\alpha_j, & \text{if } \pi_j \geq 0; \\ (1+\beta_j)^2-\pi_j^2+\alpha_j, & \text{if } -\infty < \pi_j < 0; \\ (1+\beta_j)^2+\alpha_j, & \text{if } \pi_j = -\infty. \end{cases}$$

Here β_j and π_j are the badness rating and the penalty, as before; and α_j is zero unless both line j and the previous line ended on kerfs marked as hyphenations, in which case α_j is the additional penalty assessed for consecutive hyphenated lines (e.g., 3000). We shall say that we have found the best choice of breakpoints if we have minimized the sum of δ_j over all lines j.

The above formula for δ_j is quite arbitrary, like our formula for β_j, but it works well in practice because it has the following desirable properties: (a) Minimizing the sum of squares of badnesses not only tends to minimize the badness per line, it also provides secondary optimization; for example, when one particularly bad line is inevitable, the other line breaks will also be optimized. (b) The demerit function δ_j increases as π_j increases. (c) Adding 1 to β_j instead of using the badness β_j itself will minimize the total number of lines in cases where there are several ways to break with approximately zero total badness.

For example, the total demerits charged for the paragraphs in Figures 1, 2, and 3 are 26304 for first-fit, 10491 for best-fit, and 7063 for optimum-fit. In the first-fit and best-fit methods, each line is likely to come out about as badly as any other; the optimum-fit method, by contrast, tends to have its bad cases near the beginning, since there is less flexibility in the opening lines.

FURTHER APPLICATIONS

Before we look at how an optimizing algorithm can work, let us consider in more detail how we can use the primitives of boxes, glue, and kerfs to solve a wide variety of typesetting problems. Some of these are straightforward extensions to the simple applications of the last section, while others seem at first to be quite unrelated to the ordinary task of line breaking.

Combining paragraphs

If the desired line widths are not all the same, we might want to typeset two paragraphs with the second one starting in the list of line lengths where the first one leaves off. This can be done simply by treating the two paragraphs as one, i.e., appending the second to the first, assuming that each paragraph begins with an indentation and ends with the forced paragraph-end break.

Patching

Suppose that a paragraph starts on page 100 of some book and continues on to the next page, and suppose we want to make a change to the first part of that paragraph. We want to make sure that the last line of the new page 100 will end at the right-hand margin just before the word that appears at the beginning of page 101, so that page 101 doesn't have to be redone. It is easy to specify this condition in terms of our conventions, simply by forcing a line break (using a kerf with *prebreak* and *postbreak* empty, and *join* containing an impossibly large item) at the desired place, and discarding the subsequent text. The ability of the optimum-fit algorithm to 'look ahead' means that it will find a suitable way to patch page 100 whenever it is possible to do so.

We can also force the altered part of the paragraph to have a certain number of lines, k, by using the following trick: Set the desired length l_{k+1} of the $(k+1)$st line equal to some number θ that is different from the length of any other line. Then an empty box of width θ that occurs between two forced-break items will have to be placed on line $k+1$.

Avoiding 'psychologically bad' breaks

Since computers don't know how to think, at least not yet, it is reasonable to wonder if there aren't some line breaks that a computer would choose but a human operator might not, if they somehow don't seem right. This problem does not arise very often when straight text is being set, as in newspapers or novels, but it is quite common in technical material. For example, it is psychologically bad to break before 'x' or 'y' in the sentence

A function of x is a rule that assigns a value y to every value of z.

A computer will have no qualms about breaking anywhere unless it is told not to; but a human operator might well avoid bad breaks, perhaps even unconsciously.

Psychologically bad breaks are not easy to define; we just know they are bad. When the eye journeys from the end of one line to the beginning of another, in the presence of a bad break, the second word often seems like an anticlimax, or isolated from its context. Imagine turning the page between the words 'Chapter' and '8' in some sentence; you might well think that the compositor of the book you are reading should not have broken the text at such an illogical place.

During the first year of experience with TEX, the authors began to notice occasional breaks that didn't feel quite right, although the problem wasn't felt to be severe enough to warrant corrective action. Finally, however, it became difficult to justify our claim that TEX has the world's best line-breaking algorithm, when it would occasionally make breaks that were semantically annoying; for example, the preliminary TEX manual[9] has quite a few of these, and the first drafts of that manual were even worse.

As time went on, the authors grew more and more sensitive to psychologically bad breaks, not only in the copy produced by TEX but also in other published literature, and it became desirable to test the hypothesis that computers were really to blame. Therefore a systematic investigation was made of the first 1000 line breaks in the *ACM Journal* of 1960 (which was composed manually by a Monotype operator), compared to the first 1000 line breaks in the *ACM Journal* of 1980 (which was typeset by one of the best commercially available systems for mathematics, developed by Penta Systems International). The final lines of paragraphs, and the lines preceding displays, were not considered to be line breaks, since they are forced; only the texts of articles were considered, not the bibliographies. A reader who wishes to try the same experiment should find that the 1000th break in 1960 occurred on page 67, while in 1980 it occurred on page 64. The results of this admittedly subjective procedure were a total of

13 bad breaks in 1960,
55 bad breaks in 1980.

In other words, there was more than a four-fold increase, from about 1% to a quite noticeable 5.5%! Of course, this test is not absolutely conclusive, because the style of articles in the *ACM Journal* has not remained constant, but it strongly suggests that computer typesetting causes semantic degradation when it chooses breaks solely on the basis of visual criteria.

Once this problem was identified, a systematic effort was made to purge all such breaks from the second edition of Knuth's book *Seminumerical Algorithms*[8], which was the first large book to be typeset with TEX. It is quite easy to get the line-breaking algorithm to avoid certain breaks by simply including a penalty of, say, 999 in the wordspace kerf; then the bad break is chosen only in an emergency, when there is no other good way to set the paragraph. It is possible to make the typist's job reasonably easy by reserving a special symbol (e.g., ⊗) to be used instead of a normal space between words whenever breaking is undesirable. Although this problem has rarely been discussed in the literature, the authors subsequently discovered that some typographers have a word for it: they call such spaces 'auxiliary'. Thus there is a growing awareness of the problem.

It may be useful to list the main kinds of contexts in which auxiliary spaces were used in *Seminumerical Algorithms*, since that book ranges over a wide variety of technical subjects. The following rules should prove to be helpful to compositors who are keyboarding technical manuscripts into a computer.

1. Use auxiliary spaces in cross-references:

   Theorem⊗A Algorithm⊗B Chapter⊗3 Table⊗4 Programs E and⊗F

   Note that no ⊗ appears after 'Programs' in the last example, since it would be quite all right to have 'E and F' at the beginning of a line.

2. Use auxiliary spaces between a person's forenames and between multiple surnames:

   Dr.⊗I.⊗J. Matrix Luis⊗I. Trabb⊗Pardo Peter Van⊗Emde⊗Boas

   A recent trend to avoid spaces altogether between initials may be largely a reaction against typical computer line-breaking algorithms! Note that it seems better to hyphenate a name than to break it between words; e.g., 'Don-' and 'ald E. Knuth' is more tolerable than 'Donald' and 'E. Knuth'. In a sense, rule 1 is a special case of rule 2, since we may regard 'Theorem A' as a name; another example is 'register⊗X'.

3. Use auxiliary spaces for symbols in apposition with nouns:

   base⊗b dimension⊗d function⊗$f(x)$ string⊗s of length⊗l

   However, compare the last example with 'string⊗s of length l⊗or more'.

4. Use auxiliary spaces for symbols in series:

$$1,\otimes 2,\ \text{or}\otimes 3 \qquad a,\otimes b,\ \text{and}\otimes c \qquad 1,\otimes 2,\ \ldots,\otimes n$$

5. Use auxiliary spaces for symbols as tightly-bound objects of prepositions:

$$\text{of}\otimes x \qquad \text{from 0 to}\otimes 1 \qquad \text{increase } z \text{ by}\otimes 1 \qquad \text{in common with}\otimes m$$

   This does not apply with compound objects: For example, 'of $u\otimes$and$\otimes v$'.

6. Use auxiliary spaces to avoid breaking up mathematical phrases that are rendered in words:

$$\text{equals}\otimes n \quad \text{less than}\otimes \epsilon \quad \text{mod}\otimes 2 \quad \text{modulo}\otimes p^e \quad (\text{given}\otimes X)$$

   Also type 'If $t\otimes$is ...', 'when $x\otimes$grows'. Compare 'is$\otimes$15,' with 'is 15$\otimes$times the height'; and compare 'for all large$\otimes n$' with 'for all $n\otimes$greater than$\otimes n_0$'.

7. Use auxiliary spaces when enumerating cases:

$$(\text{b})\otimes\text{Show that } f(x) \text{ is } (1)\otimes\text{continuous; } (2)\otimes\text{bounded.}$$

It would be nice to boil these seven rules down into one or two, and it would be even nicer if the rules could be automated so that keyboarding could be done without them; but subtle semantic considerations seem to be involved in many of these instances. Most examples of psychologically bad breaks seem to occur when a single symbol or a short group of symbols appears just before or after the break; one could do reasonably well with an automatic scheme if it would associate large penalties with a break just before a short non-word, and medium penalties with a break just after a short non-word. Here 'short non-word' means a sequence of symbols that is not very long, yet long enough to include instances like 'exercise$\otimes$15(b)', 'length$\otimes 2^{35}$', 'order$\otimes n/2$' followed by punctuation marks; one should not simply consider patterns that have only one or two symbols. On the other hand it is not so offensive to break before or after fairly long sequences of symbols; e.g., 'exercise 4.3.2–15' needs no auxiliary space.

Many books on composition recommend against breaking just before the final word of a paragraph, especially if that word is short; this can, of course, be done by using an auxiliary space just before that last word, and the computer could insert this automatically. Some books also give recommendations analogous to rule 2 above, saying that compositors should try not to break lines in the middle of a person's name. But there is apparently only one book that addresses the other issues of psychologically bad breaks, namely a nineteenth-century French manual by A. Frey[10], where the following examples of undesirable breaks are mentioned (vol. 1, p. 110):

$$\text{Henri}\otimes\text{IV} \qquad \text{M.}\otimes\text{Colin} \qquad 1^{\text{er}}\otimes\text{sept.} \qquad \text{art.}\otimes 25 \qquad 20\otimes\text{fr.}$$

It seems to be time to resurrect such old traditions of fine printing.

Recent experience of the authors indicates that it is not a substantial additional burden to insert auxiliary spaces when entering a manuscript into a computer. The careful use of such spaces may in fact lead to greater job satisfaction on the part of the keyboard operator, since the quality of the output can be noticeably improved with comparatively little work. It is comforting at times to know that the machine needs your help.

Author lines

Most of the review notices published in *Mathematical Reviews* are signed with the reviewer's name and address, and this information is typeset flush right, i.e., at the right-hand margin. If there is sufficient space to put such a name and address at the right of the final line of the paragraph, the publishers can save space, and at the same time the results look better because there are no strange gaps on the page (see Figure 4). During recent years the composition software used by the American Mathematical Society was unable to do this operation, but the amount of money saved on paper made it economical for them to pay someone to move the reviewer-names up by hand whenever possible, applying scissors and (real) glue to the computer output.

The author-line problem can be solved by using a kerf between the paragraph and the author's name that has the following specifications:

prebreak:	glue $(0, 10000, 0)$
postbreak:	glue $(0, 10000, 0)$
join:	glue $(w, 10000, 0)$
penalty:	p

where w is the minimum amount of white space desired between the last word of the the paragraph and the author's name, if the latter does not occur on its own line, and p is a positive penalty to encourage the algorithm to save the extra line if possible, even it it has to shrink some spaces to do it.

This is a case where the name and address fit in nicely with the review. *A. Reviewer* (Ann Arbor, Mich.)

But sometimes an extra line must be added.
N. Bourbaki (Paris)

Figure 4. The author-line problem.

Ragged margins

In the previous section we discussed how to produce ragged right margins. Occasionally it is also desirable to be able to typeset ragged left margins; this can be done in essentially the same way as ragged right margins, except that the stretchy glue is moved from the *prebreak* specification to *postbreak*. Also, the paragraph should be started with a glue item with the same stretchability as in the *postbreak*, and the stretchy glue should be taken out of the forced break at the end of the paragraph. Notice that this will set the last line of the paragraph to be roughly the same size as the rest; this is probably what is wanted for ragged-left text.

If the stretchy glue is included in both *prebreak* and *postbreak*, a 'ragged centered' setting is obtained, which might be useful on a title page or poster.

These constructions for ragged margins essentially minimize the largest gap between the margins and the text on any line; and subject to that minimum they essentially minimize the largest gap on the remaining lines; and so forth. The reason is that our definitions of 'badness' and 'demerits' imply in this case that the sum of demerits for any choice of breakpoints is approximately proportional to the sixth powers of the individual gaps.

Hyphenation of ligatures

Early versions of TEX could not allow discretionary hyphens in ligatures because the model for line breaking was originally formulated with a less general concept than kerfs. Later, each of the authors independently invented a variety of kerf, primarily to allow just these hyphenations.

For example, if the ligature 'ff' occurs in a word such that it is legal to hyphenate between the two letters, it would be represented by a kerf with *prebreak* containing 'f-', *postbreak* containing 'f', *join* containing 'ff', *hyph* true, and *penalty* as appropriate for a hyphenation.

Further applications

The reader who is interested in seeing even more complicated applications of the line breaking algorithm, applied to the typesetting of computer programs and to entries in a sophisticated index, is referred to the earlier paper on line breaking by the present authors[11]. That paper predates the notion of kerfs, but it is not difficult to reformulate those solutions in terms of generalized kerfs.

Description of the Algorithm

The general ideas underlying the optimum-fit algorithm for line breaking can probably be understood best by considering an example. Figure 5 repeats the text of the earlier figures, set at a different width. It includes little vertical marks to indicate 'feasible breakpoints' found by the algorithm. A *feasible breakpoint* is a place where the text of the paragraph from the beginning to this point can be broken into lines whose badness does not exceed a given tolerance; in the case of Figure 5, this tolerance was taken to be 100. Thus, for example, there is a tiny mark after 'fountain;' since there is a way to set the paragraph up to this point with 'fountain;' at the end of the 7th line and with none of lines 1 to 7 having an adjustment ratio exceeding unity.

The algorithm proceeds by locating all of the feasible breakpoints and remembering the best way to get to each one, in the sense of fewest total demerits. This is done by keeping a list of 'active' breakpoints, representing all of the feasible breakpoints that might be a candidate for future breaks. Whenever a potential breakpoint b is encountered, the algorithm tests to see if there is any active breakpoint a such that the line from a to b has an acceptable badness. If so, b is a feasible breakpoint and it is appended to the active list. The algorithm also remembers the identity of the breakpoint a that minimizes the total demerits, when the total is computed from the beginning of the paragraph to b through a. When an active breakpoint a is encountered for which the line from a to b has an adjustment ratio less than -1 (i.e., when the line can't be shrunk to fit the desired length), breakpoint a is removed from the active list. Since the size of the active list is essentially bounded by the maximum number of words per line, the running time of the algorithm is bounded by this quantity (which usually is small) times the number of potential breakpoints.

For example, when the algorithm begins to work on the paragraph in Figure 5, there is only one active breakpoint, representing the beginning of the first line. It is infeasible to have a line starting there and ending at 'In', or 'olden', ..., or 'lived', since the glue between words does not accumulate enough stretchability in such short segments of the text; but after the next word 'a' is encountered, a feasible breakpoint is found. Now there are two active breakpoints, the original one and the new one. After the next word 'king', there are three active breakpoints; but after the next word 'whose', the algorithm sees that it is impossible to squeeze all of the text from the beginning up to 'whose' on one line, so the initial breakpoint becomes inactive and only two active ones remain.

Skipping ahead, let us consider what happens when the algorithm considers the potential break after 'fountain;'. At this stage there are eight active breakpoints, following the respective text boxes for 'child', 'went', 'out', 'side', 'of', 'the', 'cool', and 'foun-'. The line starting after 'child' and ending with 'fountain;' would be too long to fit, so 'child' becomes inactive. Feasible lines are found from 'went' or 'out' to 'fountain;' and the demerits of those lines are 276 and 182, respectively; however, the line from 'went' actually turns out to be preferable, since there are substantially fewer total demerits from the beginning of the paragraph to 'went' than to

Figure 5. A somewhat wider setting of the same sample paragraph, by the optimum-fit method. Tiny vertical marks show 'feasible breakpoints' where it is possible to break in such a way that no spaces need to stretch more than their given stretchability.

In olden times when wishing still helped one, there lived a king whose daughters were all beautiful; and the youngest was so beautiful that the sun itself, which has seen so much, was astonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and under an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain; and when she was bored she took a golden ball, and threw it up on high and caught it; and this ball was her favorite plaything.

'out'. Thus, 'fountain;' becomes a new active breakpoint. The algorithm stores a pointer back from 'fountain;' to 'went', meaning that the best way to get to a break after 'fountain;' is to start with the best way to get to a break after 'went'.

The computation of this algorithm can be represented pictorially by means of the network in Figure 6, which shows all of the feasible breakpoints together with the number of demerits charged for each feasible line between them. The object of the algorithm is to compute the *shortest path* from the top of Figure 6 to the bottom, using the demerit numbers as the 'distances' corresponding to individual parts of the path. In this sense, the job of optimal line breaking is essentially a special case of the problem of finding shortest paths in an acyclic network; the line-breaking algorithm is slightly more complex only because it must construct the network at the same time as it is finding the shortest path.

Notice that the best-fit algorithm can be described very easily in terms of a network like Figure 6: it is the algorithm that simply chooses the shortest continuation at every step. And the first-fit algorithm can be characterized as the method of always taking the leftmost branch having a negative adjustment ratio (unless it leads to a hyphen, in which case the rightmost non-hyphenated branch is chosen whenever there is a feasible one). From these considerations we can readily understand why the optimum-fit algorithm tends to do a much better job.

Sometimes there is no way to continue from one feasible breakpoint to any other. This situation doesn't occur in Figure 6, but it would be present below the word 'so' if we had not permitted hyphenation of 'astonished'. In such cases the first-fit and best-fit algorithms must resort to infeasible lines, while the optimum-fit algorithm can usually find another way through the maze.

On the other hand, some paragraphs are inherently difficult, and there is no way to break them into feasible lines. In such cases the algorithm we have described will find that its active list dwindles until eventually there is no activity left; what should be done in such a case? It would be possible to start over with a more tolerant attitude toward infeasibility (a higher threshold value for the badness). Alternatively, TEX takes the attitude that the user wants to make some manual adjustment when there is no way to meet the specified criteria, so the active list is forcibly prevented from becoming empty by simply declaring a breakpoint to be feasible if it would otherwise leave the active list empty. This results in an overset line and an error message that encourages the user to take corrective action.

Figure 7 shows what happens when the algorithm allows quite loose lines to be feasible; in this case a line is considered to be infeasible only if its adjustment ratio exceeds 10 (so that there would be more than two ems of space between words). Such a setting of the tolerances would be used by people who don't want to make manual adjustments to paragraphs that cannot be set well. The tiny marks that indicate feasible breakpoints have varying lengths in this illustration, with longer marks indicating places that can be reached via better paths; the tiny dots are for breakpoints that are just barely feasible. Notice that all of the potential breakpoints in Figure 7 are marked, except for a few in the first two lines; so there are considerably more feasible

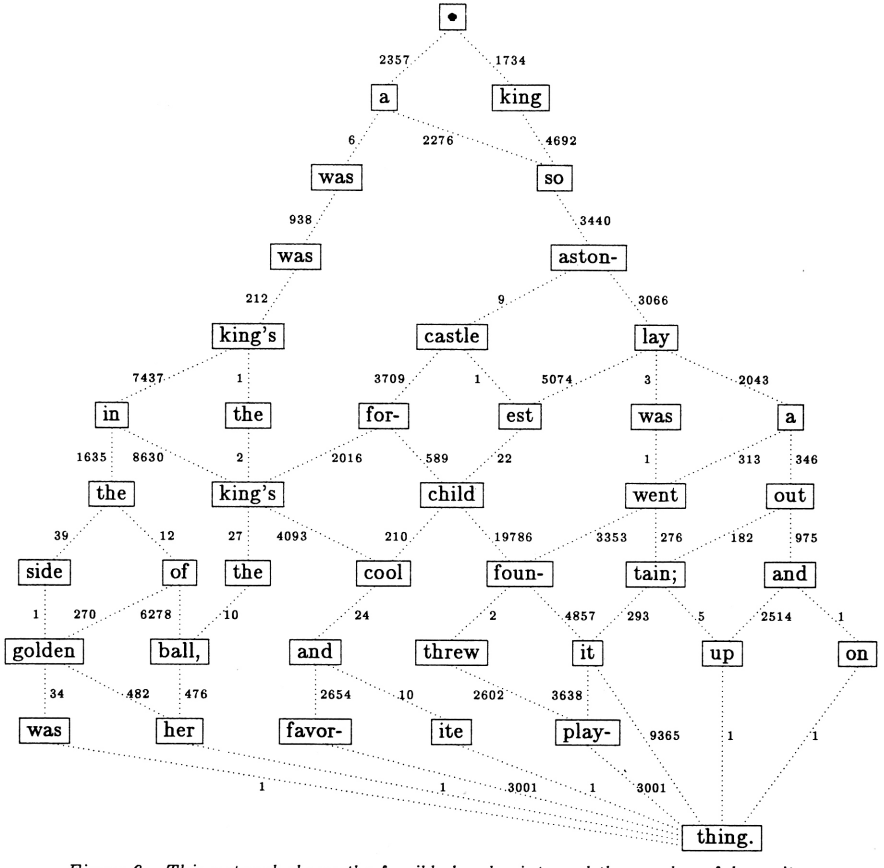

Figure 6. This network shows the feasible breakpoints and the number of demerits charged when going from one breakpoint to another. The 'shortest path' from the top to the bottom corresponds to the best way to typeset the paragraph, if we regard the demerits as distances.

breakpoints here than there were in Figure 5, and the network corresponding to Figure 7 will be much larger. There are 836,272,858 feasible ways to set the paragraph when such wide spaces are tolerated, compared to only 81 ways in Figure 5. However, the number of active nodes will not be significantly bigger in this case than it was in Figure 5, because it is limited by the length of a line, so the algorithm will not run too much more slowly even though its tolerance has been raised and the number of possible settings has increased enormously. For example, after 'fountain;' there are now 17 active breakpoints instead of the 8 present before, so the processing takes only about twice as long although huge numbers of additional possibilities are being taken into account.

When the threshold allows wide spacing, the algorithm is almost certain to find a feasible solution, and it will report no errors to the user even though some rather loose lines may have been necessary. The user who wants such error messages should set the tolerance lower; this not only gives warnings when corrective action is needed, it also improves the algorithm's efficiency.

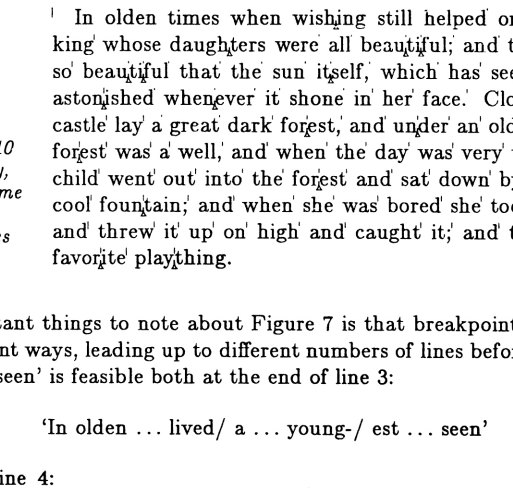
In olden times when wishing still helped one, there lived a ...0
king whose daughters were all beautiful; and the youngest was ...4
so beautiful that the sun itself, which has seen so much, was ...6
astonished whenever it shone in her face. Close by the king's ...4
castle lay a great dark forest, and under an old lime-tree in the ...7
forest was a well, and when the day was very warm, the king's ...5
child went out into the forest and sat down by the side of the ...4
cool fountain; and when she was bored she took a golden ball, ...5
and threw it up on high and caught it; and this ball was her ...3
favorite plaything. ...2

Figure 7. When the tolerance is raised to 10 times the stretchability, more breakpoints become feasible, and there are many more possibilities to explore.

One of the important things to note about Figure 7 is that breakpoints can become feasible in completely different ways, leading up to different numbers of lines before the breakpoint. For example, the word 'seen' is feasible both at the end of line 3:

'In olden ... lived/ a ... young-/ est ... seen'

and at the end of line 4:

'In olden ... helped/ one, ... were/ all ... beau-/ tiful ... seen',

although 'seen' was not a feasible break at all in Figure 5. The breaks that put 'seen' at the end of line 3 have substantially fewer demerits than those putting it on line 4 (approximately 1.68×10^6 versus 1.28×10^{10}), so the algorithm will remember only the former possibility. This is an application of the dynamic-programming 'principle of optimality', which is responsible for the efficiency of our algorithm[4]: the optimum breakpoints of a paragraph are always optimum for the subparagraphs they create. But the interesting thing is that this economy of storage would not be possible if the future lines were not all of the same length, since differing line lengths might well mean that it would be much better to put 'seen' on line 4 after all; for example, we have mentioned a trick for forcing the algorithm to produce a given number of lines. In the presence of varying line lengths, therefore, the algorithm would need to have two separate list entries for an active breakpoint after the word 'seen'. The computer cannot simply remember the one with fewest total demerits, because the optimality principle of dynamic programming would not be valid in such a case.

Figure 8 is an example of line breaking when the individual lengths are all different. In such cases, the need to attach line numbers to breakpoints might mean that the number of active breakpoints substantially exceeds the maximum number of words per line, if the feasibility tolerance is set high; so it is desirable to set the tolerance low. On the other hand, if the tolerance is set too low, there may be no way to break the paragraph into lines having a desired shape. Fortunately, there is usually a happy medium in which the algorithm has enough flexibility to find a good solution without needing too much time and space. The data in Figure 9 shows, for example, that the algorithm did not have to do very much work to find an optimal solution for Galileo's remarks on circles, when the adjustment ratio on each feasible line was required to be 2 or less; yet there was sufficient flexibility to make feasible solutions possible.

COMPUTATIONAL EXPERIENCE

The TEX typesetting system uses an algorithm similar to the one described in the previous section. Since TEX must deal with other aspects of typesetting, such as leaders, lines with attached footnotes or cross references or page-break marks, and with spacing both inside and immediately outside of math formulas, the implementation of the line breaking algorithm

The area of a
circle is a mean propor-
tional between any two regular
and similar polygons of which one
circumscribes it and the other is iso-
perimetric with it. In addition, the area
of the circle is less than that of any cir-
cumscribed polygon and greater than that
of any isoperimetric polygon. And further,
of these circumscribed polygons, the one
that has the greater number of sides has
a smaller area than the one that has
a lesser number; but, on the other
hand, the isoperimetric polygon
that has the greater num-
ber of sides is the
larger.

— Galileo Galilei (1638)

I
turn, in the
following treatises, to
various uses of those triangles
whose generator is unity. But I leave out
many more than I include; it is extraordinary how
fertile in properties this triangle is. Everyone can try his hand.

— Blaise Pascal (1654)

Figure 8. Examples of line breaking with lines of different sizes.

'The area of a' .000
circle is a mean propor-' .001
tional between any two regular' .021
and similar polygons of which one' .044
circumscribes it and the other is iso-' .208
perimetric with it. In addition, the area' .007
of the' circle is less than that of any cir-' .067
cumscribed' polygon and greater than that' .073
of any' isoperimetric polygon. And further,' .000
of these' circumscribed polygons, the one' .001
that' has' the' greater number of sides has' .720
a' smaller' area' than the one that has' .425
a' lesser number;' but, on the other' .288
hand,' the' isoperimetric polygon' .163
that has the' greater num-' .000
ber' of sides is' the' .000
larger. .000

Figure 9. Details of the feasible breakpoints in the first example of Figure 8, showing how the optimum solution was found.

is quite complex. (A complete description of TEX's algorithm will appear elsewhere[12].) Experience has shown that the general algorithm is quite efficient in practice, in spite of all the things it must cope with.

So many parameters are present, it is impossible for anyone actually to experiment with a large fraction of the possibilities. A user can vary the interword spacing and the penalties for inserted hyphens, explicit hyphens, and adjacent flagged lines. In addition, TEX's algorithm has some more refinements we have not covered here, each with its own parameters: An extra penalty can be charged whenever a very tight line is next to a loose one, or a very loose line is next to a normal one. TEX's approach for hyphenation is to first try setting the paragraph without any hyphenations, and then only if a certain tolerance is exceeded are the discretionary hyphens inserted. In the presence of so many parameters, one could perform computational experiments for years and not have a completely definitive idea about the behavior of this algorithm. Even with fixed parameters there is a significant variation with respect to the kind of material being typeset; for example, highly mathematical copy presents special problems. An interesting comparative study of line breaking was made by Duncan et al.[7], who considered sample texts from Gibbon's *Decline and Fall* versus excerpts from a story entitled *Salar the Salmon*; as expected, Gibbon's vocabulary forced substantially more hyphenated lines.

On the other hand, we have seen that the optimizing algorithm leads to better line breaks even in children's stories like Grimm's fairy tales, where the words are short and simple. It would be nice to have a quantitative feeling for how much extra computation is necessary to get this improvement in quality. Roughly speaking, the computation time is proportional to the number of words of the paragraph, times the average number of words per line, since the main loop of the computation runs through the currently active nodes, and since the average number of words per line is a reasonable estimate of the number of active nodes in all but the first few lines of a paragraph (see Figures 5 and 7). On the other hand, there are comparatively few active nodes on the first lines of a paragraph, so the performance is actually faster than this rough estimate would indicate.

Detailed statistics were kept when TEX's first large production, *Seminumerical Algorithms*[8], was typeset using the above procedure. This 700-page book has a total of 5526 'paragraphs' in its text and answer pages, if we regard displayed formulas as separators between independent paragraphs. The 5526 paragraphs were broken into a total of 21,057 lines, of which 550 (about 2.6 per cent) ended with hyphens. The lines were usually 29 picas wide, which means 626.4 machine units in 10-point type and about 677.19 machine units in 9-point type, roughly twelve or thirteen words per line. The threshold values were normally both set to 200, so that the spaces between words ranged from a minimum of 4 units to a maximum of $6 + 3\sqrt[3]{2} \approx 9.78$ units. The penalty for breaking after a hyphen was 50, and 3000 demerits were charged for consecutive hyphens and for adjacent lines with large adjustment ratio changes. The second (hyphenation) pass was needed on only 279 of the paragraphs, i.e., about 5% of the time; a feasible solution without hyphenation was found in the remaining 5247 cases. The second pass would only try to hyphenate uncapitalized words of five or more letters, containing no accents, ligatures, or hyphens, and it turned out that exactly 6700 words were submitted to the hyphenation procedure. Thus the number of attempted hyphenations per paragraph was approximately 1.2, only slightly more than needed by conventional nonoptimizing algorithms, and this was not a significant factor in the running time.

The main contribution to the running time came, of course, from the main loop of the algorithm, which was executed 274,102 times (about 50 times per paragraph, including both passes lumped together when the second pass was needed). The total number of break nodes created was 64,003 (about 12 per paragraph), including multiplicities for the comparatively rare cases that different fitness classifications or line numbers needed to be distinguished for the same breakpoint. Thus, about 23% of the legal breakpoints turned out to be feasible ones, given these comparatively low values of the tolerances. The inner loop of the computation

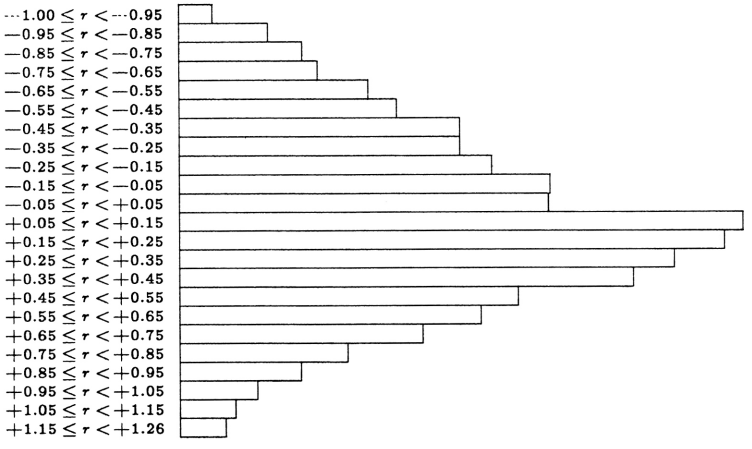

Figure 10. The adjustment ratios for interword spaces in a 700-page book.

was performed 880,677 times; this is the total number of active nodes examined when each legal breakpoint was processed, summed over all legal breakpoints. Note that this amounts to about 160 active node examinations per paragraph, and 3.2 per breakpoint, so the inner loop definitely dominates the running time. If we assume that words are about five letters long, so that a legal break occurs for every six characters of input text including the spaces between words, the algorithm costs about half of an inner-loop step per character of input, plus the time to pass over that character in the outermost loop.

And how about the output? Figure 10 shows the actual distribution of adjustment ratios r in the 15,531 typeset lines of *Seminumerical Algorithms*, not counting the 5526 lines at the ends of paragraphs, for which $r \approx 0$. There was also one line with $r \approx 1.8$ and one with $r \approx 2.2$ (i.e., a disgraceful spacing of 12.6 units); perhaps some reader will be able to spot one or both of these anomalies some day. The average value of r over all 21,057 lines was 0.08, and the standard deviation was only 0.403; about 67% of the lines had word spaces varying between 5 and 7 units. Furthermore the author believes that virtually none of the 15,531 line breaks are 'psychologically bad' in the sense mentioned above.

Anyone who has experience with typical English text knows that these statistics are not only excellent, they are in fact too good to be true; no line-breaking algorithm can achieve such stellar behavior without occasional assists from the author, who notices that a slight change in wording will permit nicer breaks. Indeed, this is another source of improved quality when an author is given composition tools like TEX to work with, because a professional compositor does not dare mess around with the given wording when setting a paragraph, while an author is happy to make changes that look better, especially when such changes are negligible by comparison with changes that are found to be necessary for other reasons when a draft is being proofread. An author knows that there are many ways to say what he or she wants to say, so it is no trick at all to make an occasional change of wording.

Theodore L. De Vinne, one of America's foremost typographers at the turn of the century, wrote[13] that 'when the author objects to [a hyphenation] he should be asked to add or cancel or substitute a word or words that will prevent the breakage ... Authors who insist on even spacing always, with sightly divisions always, do not clearly understand the rigidity of types.' Another interesting comment was made by G. B. Shaw[14]: 'In his own works, whenever [William Morris]

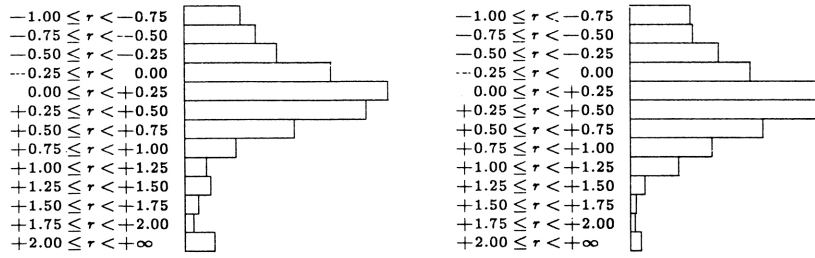

Figure 11. The distribution of interword spaces found by the best line-at-a-time method, compared to the distribution found by the best paragraph-at-a-time method, when difficult mathematical copy is typeset without human intervention.

found a line that justified awkwardly, he altered the wording solely for the sake of making it look well in print. When a proof has been sent to me with two or three lines so widely spaced as to make a grey band across the page, I have often rewritten the passage so as to fill up the lines better; but I am sorry to say that my object has generally been so little understood that the compositor has spoilt all the rest of the paragraph instead of mending his former bad work.'

The bias caused by Knuth's tuning his manuscript to a particular line width makes the statistics in Figure 10 inapplicable to the printer's situation where a given text must be typeset as it is. So another experiment was conducted in which the material of Section 3.5 of *Seminumerical Algorithms* was set with lines 25 picas wide instead of 29 picas. This sample text, which deals with the question 'What is a random sequence?', was chosen because it most closely resembles typical mathematics papers containing theorems, proofs, lemmas, etc. The optimum-fit algorithm now had to work harder than it did when the material was set to 29 picas, primarily because the second pass was needed about thrice as often (49 times out of 273 paragraphs, instead of 16 times); furthermore the second pass was much more tolerant of wide spaces in order to guarantee that every paragraph could be typeset without manual intervention. There were about 6 examinations of active nodes per legal breakpoint encountered, instead of about 3, so the net effect of this change in parameters was to nearly double the running time for line breaking. The reason for such a discrepancy was primarily the combination of difficult mathematical copy and a narrower column measure, rather than the 'author tuning', because when the same text was set 35 picas wide the second pass was needed only 8 times.

It is interesting to observe the quality of the spacing obtained in this 25-pica experiment, since it indicates how well the optimum-fit method can do without any human intervention. Figure 11 shows what was obtained, together with the corresponding statistics for the best-fit method when it was applied to the same data. About 800 line breaks were involved in each case, not counting the final lines of paragraphs. The main difference was that optimum-fit tended to put more lines into the range $.5 \leq r \leq 1$, while best-fit produced considerably more lines that were extremely spaced out. The standard deviation of spacing was 0.53 (optimum-fit) versus 0.65 (best-fit); 24 of the lines typeset by best-fit had spaces exceeding 12 units, while only 7 such bad lines were produced by the optimum-fit method. An examination of these seven problematical cases showed that three of them were due to long unbreakable formulas embedded in the text, three were due to the rule that TEX does not try to hyphenate capitalized words, and the other one was due to TEX's inability to hyphenate the word 'reasonable'. Cursory inspection of the output indicated that the main difference between best-fit and optimum-fit, in the eyes of a casual reader, would be that the best-fit method not only resorted to occasional wide spacing, it also tended to end substantially more lines with hyphens: 119 by comparison

with 80. An author who cares about spacing, and who therefore will edit a manuscript until it can be typeset satisfactorily, would have to do a significant amount of extra work in order to get the best-fit method to produce decent results with such difficult copy, but the output of the optimum-fit method could be made suitable with only a few author's alterations.

Problems and Refinements

Books on typography frequently discuss a problem that may be the most serious consequence of loose typesetting, the occasional gaps of white space that are called 'houndsteeth' or 'lizards' or 'rivers.' Such ugly patterns, which run up through a sequence of lines and distract the reader's eye, cannot be eliminated by a simple efficient technique like dynamic programming. Fortunately, however, the problem almost never arises when the optimum-fit algorithm is used, because the computer is generally able to find a way to set the lines with suitably tight spacing. Rivers begin to be prevalent only when the tolerance threshold has been set high for some reason, for example when an unusually narrow column is being justified. Another case that sometimes leads to rivers arises when the text of a paragraph falls into a strictly mechanical pattern, as when a newspaper lists all of the guests at a large dinner party. Extensive experience with TEX has shown, however, that manual removal of rivers is almost never necessary after the optimum-fit algorithm has been used.

The box-glue-kerf model applies in the vertical dimension as well as in the horizontal, so TEX is able to make fairly intelligent decisions about where to start each new page. The tricks we have discussed for such things as ragged-right setting correspond to analogous vertical tricks for such things as 'ragged-bottom' setting. However, the current implementation of TEX keeps each page in memory until it has been output, so TEX cannot store an entire document and find strictly optimum page breaks using the algorithm we have presented for line breaks. The 'best-fit' method is therefore used to output one page at a time.

Experiments have been made with a two-pass version of TEX that does find globally optimum page breaks. This experimental system also helps with the positioning of illustrations as near as possible to where they are cited, and it addresses the difficult layout problems posed by extensive footnotes in certain scholarly works. Many of these issues can be resolved by extending the dynamic programming technique and the model of this paper, but some closely related problems can be shown to be NP-complete[15].

REFERENCES

[1] Michael P. Barnett, *Computer Typesetting: Experiments and Prospects*, M.I.T. Press, Cambridge, Mass., 1965.

[2] C. J. Duncan, "Look! No hands!,' *The Penrose Annual* **57**, 121–168 (1964).

[3] Michael R. Garey and David S. Johnson, *Computers and Intractability*, W. H. Freeman, San Francisco, 1979.

[4] Richard Bellman, *Dynamic Programming*, Princeton Univ. Press, Princeton, N.J., 1957.

[5] M. Held and R. M. Karp, 'The construction of discrete dynamic programming algorithms,' *IBM Systems J.* **4**, 136–147 (1965).

[6] Jakob Ludwig Karl Grimm and Wilhelm Karl Grimm; 'Der Froschkönig (The Frog King),' in *Kinder- und Hausmärchen*, first published in Berlin, 1812. For the history of this story see Heinz Rölleke, *Die Altese Märchensammlung der Brüder Grimm*, Fondation Martin Bodmer, Cologny-Genève, 1975, pp. 144–153.

[7] C. J. Duncan, J. Eve, L. Molyneux, E. S. Page, and Margaret G. Robson, 'Computer typesetting: an evaluation of the problems,' *Printing Technology* **7**, 133–151 (1963).

[8] Donald E. Knuth, *Seminumerical Algorithms*, Vol. 2 of *The Art of Computer Programming*, second edition, Addison–Wesley, Reading, Massachusetts, 1981.

[9] Donald E. Knuth, *TEX and METAFONT: New Directions in Typesetting*, American Mathematical Society and Digital Press, Bedford, Massachusetts, 1979.

[10] A. Frey, *Manuel Nouveau de Typographie*, Paris (1835), 2 vols.

[11] Donald E. Knuth and Michael F. Plass, 'Breaking paragraphs into lines,' *Software—Practice and Experience*, **11**, 1119–1184 (1981).

[12] Donald E. Knuth, *Tau Epsilon Chi: A System for Technical Text*, book in preparation.

[13] Theodore Low De Vinne, *Correct Composition*, Vol. 2 of *The Practice of Typography*, Century, New York, 1901. The cited material appears on pages 138 and 206.

[14] George Bernard Shaw, 'On Modern Typography,' *The Dolphin* **4**, 80–81 (1940).

[15] Michael F. Plass, 'Optimal pagination techniques for automatic typesetting systems,' Ph.D. thesis, Stanford University, June 1981.